RDFAlchemy Documentation

Release 0.2b3

Philip Cooper, Graham Higgins

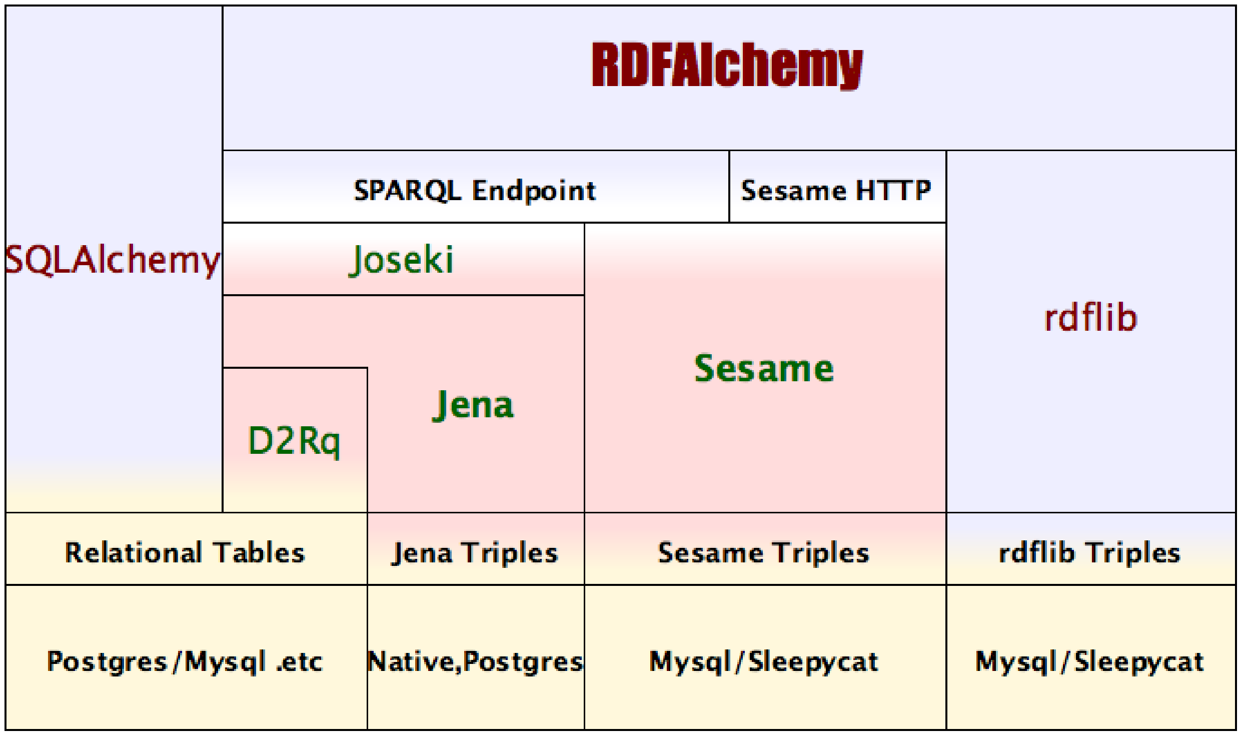
January 16, 2012

CONTENTS

The goal of RDF Alchemy is to allow anyone who uses python to have a object type API access to an RDF Triplestore.

The same way that:

- SQLAlchemy is an ORM (Object Relational Mapper) for relational database users
- RDFAlchemy is an ORM (Object RDF Mapper) for semantic web users.



The use of persistent objects in RDFAlchemy will be as close as possible to what it would be in SQLAlchemy. Code like:

```
>>> c = Company.query.get_by(symbol = 'IBM')
>>> print(c.companyName)
International Business Machines Corp.
```

This code does not change as the user migrates from SQLAlchemy to RDFAlchemy and back, lowering the bar for adoption of RDF based datastores.

Contents:

CHAPTER

ONE

RDFALCHEMY ORM

1.1 rdfSubject

rdfalchemy.py - a Simple API for RDF

Requires rdflib <http://www.rdflib.net/> version 2.3 ??.

class `rdfalchemy.rdfSubject.`**`rdfSubject`**(*resUri=None*, ***kwargs*)

classmethod **`ClassInstances`**()

return a generator for instances of this rdf:type you can look in MyClass.rdf_type to see the predicate being used

classmethod **`GetRandom`**()

for develoment just returns a random instance of this class

`db` **= <Graph identifier=AdDfxahH0 (<class 'rdflib.graph.ConjunctiveGraph'>)>**

Default graph for access to instances of this type

classmethod **`filter_by`**(***kwargs*)

Class method returns a generator over classs instances meeting the kwargs conditions.

Each keyword must be a class descriptor

filter by RDF.type == cls.rdf_type is implicit

Order helps, the first keyword should be the most restrictive

classmethod **`get_by`**(***kwargs*)

Class Method, returns a single instance of the class by a single kwarg. the keyword must be a descriptor of the class. example:

```
bigBlue = Company.get_by(symbol='IBM')
```

Note the keyword should map to an rdf predicate that is of type owl:InverseFunctional

`md5_term_hash`()

Not sure what good this method is but it's defined for rdflib.Identifiers so it's here for now

`n3`()

n3 repr of this node

`rdf_type` **= None**

rdf:type of instances of this class

1.2 rdfsSubject

rdfsSubject.py

rdfsSubject is similar to rdfsSubject but includes more processing and *magic* based on an RDF Schema

Created by Philip Cooper on 2008-05-14. Copyright (c) 2008 Openvest. All rights reserved.

class `rdfalchemy.rdfsSubject.`**`rdfsSubject`**(*resUri=None*, ***kwargs*)

> **classmethod** **`ClassInstances`**()
>
> return a generator for instances of this rdf:type you can look in MyClass.rdf_type to see the predicate being used

1.3 Descriptors

Understanding Descriptors is key to using RDFAlchemy. A descriptor binds an instance variable to the calls to the RDF backend storage.

Class definitions are simple with the RDFAlchemy Descriptors. The descriptors are implemented with caching along the lines of this recipe. The predicate must be passed in.

```python
ov = Namespace('http://owl.openvest.org/2005/10/Portfolio#')
vcard = Namespace("http://www.w3.org/2006/vcard/ns#")

class Company(rdfSubject):
    rdf_type = ov.Company
    symbol = rdfSingle(ov.symbol,'symbol')  #second param is optional
    cik = rdfSingle(ov.secCik)
    companyName = rdfSingle(ov.companyName)
    address = rdfSingle(vcard.adr)
    stock = rdfMultiple(ov.hasIssue)

c = Company.query.get_by()
print("%s has an SEC symbol of %s" % (c.companyName, c.cik))
```

Note: In addition to the `get_by`, which returns a single instance, there is now a `filter_by` which returns a list of instances.

1.3.1 The RDFAlchemy Descriptors

`rdfalchemy.descriptors.`**`rdfSingle`**(*pred*, *cacheName=None*, *range_type=None*)

> This is a Descriptor Takes a the URI of the predicate at initialization Expects to return a single item on Assignment will set that value to the ONLY triple with that subject,predicate pair

`rdfalchemy.descriptors.`**`rdfMultiple`**(*pred*, *cacheName=None*, *range_type=None*)

> This is a Descriptor Expects to return a list of values (could be a list of one)

`rdfalchemy.descriptors.`**`rdfList`**(*pred*, *cacheName=None*, *range_type=None*)

> This is a Descriptor Expects to return a list of values (could be a list of one) *__set__* will set the predicate as a RDF List

`rdfalchemy.descriptors.`**`rdfContainer`**(*pred*, *range_type=None*, *container_type='http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq'*)

This is a Descriptor Expects to return a list of values (could be a list of one)

container_type in __*init*__ should be one of

- rdf:Seq
- rdf:Bag
- rdf:Alt

__*set*__ will set the predicate as a RDF Container type (defaults to rdf:Seq)

`rdfalchemy.descriptors.`**`owlTransitive`**(*pred*, *cacheName=None*, *range_type=None*)

owlTransitive is a descriptor based on a transitive predicate The predicate should be of type owl:TransitiveProperty

1.3.2 Returned values

- `rdfSingle` returns a single literal
- `rdfMultiple` returns a list (may be a list of one)
- **`rdfMultiple` will return a python list if the predicate is:**
  - in multiple triples for the `(s p o1)` `(s p o2)` etc yields `o2 <o1,>` _
  - points to an RDF Collection (`rdf:List`)
  - points to an RDF Container (`rdf:Seq`, `rdf:Bag` or `rdf:Alt`)
- `rdfList` returns a list (may be a list of one) and on save will save as an `rdf:Collection` (aka List)
- `rdfContainer` returns a list and on save will save as an `rdf:Seq`.

1.3.3 Chained descriptors

The `__init__` functions for the Descriptors now takes an optional argument of `range_type`. If you know the rdf.type (meaning the uriref of the type) you may pass it to the `Class.__init__`.

Within the samples module, a `DOAP.Project` maintainer is a `FOAF.Person`

```
DOAP=Namespace("http://usefulinc.com/ns/doap#")
FOAF=Namespace("http://xmlns.com/foaf/0.1/" )

class Project(rdfSubject):
    rdf_type = DOAP.Project
    name = rdfSingle(DOAP.name)
    # ... some other descriptors here
    maintainer = rdfSingle(DOAP.maintainer,range_type=FOAF.Person)

from rdfalchemy.samples.foaf import Person
from rdfalchemy.orm import mapper

mapper()
# some method to find an instance
p = Doap.ClassInstances().next()
p.maintainer.mbox
```

To get such mapping requires 3 steps:

1. Classes must be declared with the proper *rdf_type* Class variable set
2. Descriptors that return an instance of a python class should be created with the optional parameter of range_type with the same type as in step 1.
3. Call the `mapper()` function from *rdfalchemy.orm*. This can be called later to 'remap' classes at any time.

The bindings are not created until the third step so classes and descriptors can be created in any order.

1.3.4 Chained predicates

Predicates can now be chained as in

```
c = Company.query.get_by(symbol='IBM')
print(c[vcard.adr][vcard.region])
## or
print(c.address[vcard.region])
```

This works because the generic `rdfSubject[predicate.uri]` notation maps to `rdfSubject.__getitem__` which endeavors to return an instance of `rdfSubject`.

1.4 Mapper

`rdfalchemy.orm.`**`mapper`**`(`*`*classes`*`)`

Maps the classes given to allow descriptors with ranges to the proper Class of that type

The default, if no args are provided, is to map recursively all subclasses of `rdfSubject`

Returns a dict of {rdf_type: mapped_class} for further processing

`rdfalchemy.orm.`**`allsub`**`(`*`cl, beenthere=set([])`*`)`

return all subclasses of the given class

1.5 Hybrid SQL/RDF Alchemy Objects

If we look at the requirements for any python based object to respond to RDFAlchemy requests there are only two requirements:

1. That some instance object `inst` be able to respond to an `inst.resUri` call (it needs to know its URI) 2. That there be some descriptor (like `rdfSingle`) defined for the instance `obj` or its class `type(obj)`

The first requirement could be satisfied by creating some type of mixin class and inheriting from multiple base objects. Maybe I'll go there some day but the behavior of `get_by` would be uncertain (unless I reread the precedence rules :-). In the mean time we can assign or lookup the relevant URI for the object (assignment could be defined via the D2Rq vocabulary).

From there you can assign descriptors on the fly and access your Triplestore. RDFDescriptors pull from the RDF Triplestore like rdf via RDFAlchemy and the rest pull from the relational database via SQLAlchemy. A developer need not put all of his data in one repository.

You can mix and match SQL, RDF and SPARQL data with little effort.

CHAPTER

TWO

CRUD

2.1 Create

```
class Person(rdfSubject):
        rdf_type = FOAF.Person
        first = rdfSingle(FOAF.givenname)
        last = rdfSingle(FOAF.surname)

p1 = Person() # creates a bnode with an 'foaf:Person <rdf:type>'_ triple
p2 = Person('<http://www.openvest.com/user/phil') #creates a URIRef with the same triple
p3 = Person(last="Cooper",first="Philip") #creates a bnode with 3 triples (rdf:type FOAF:surname FOAI
```

2.2 Read

Reading is simply a matter of using the declared descriptors

```
c = Company.query.get_by(symbol = 'IBM')
print(c.companyName)
print(c.address.region)
```

If a descriptor is not defined for a predicate and you still want to access the value you can use the `__getitem__` dictionary type access

```
print(c[ov.companyName])
print(c[vcard.adr][vcard.region])
```

The flexibility of the item access is ok but descriptors should be used whenever possible as they are much more intelligent. They:

- cache database calls
- return the proper class of the returned item if `mapper()` has been called
- return lists correctly for collections (Lists, and Containers both)

2.3 Update

Writing to the database for rdflib is done at the time of assignment. It currently only performs `set` or `delete` operations for `rdfSingle` descriptors as the behavior for `rdfMultiple` is more ambiguous.

The basic syntax for the `rdfSingle` descriptors is:

```
ibm Company.query.get_by(symbol = 'IBM')
sun Company.query.get_by(symbol = 'JAVA')

## add another descriptor on the fly
Company.industry = rdfSingle(ov.yindustry,'industry')

## add an attribute (to the database)
sun.industry = 'Computer stuff'

## delete an attribute (from the database)
del ibm.industry
```

2.4 Delete

To delete a record, use the `remove()` method. Removing an object from a graph database is more complicated than removing the triples where the item is the subject of the triple.

```
def remove(self, node=None, db=None, cascade = 'bnode', bnodeCheck=True):
        """remove all triples where this rdfSubject is the subject of the triple
        db -- limit the remove operation to this graph
        node -- node to remove from the graph defaults to self
        cascade -- must be one of:
                    * none -- remove none
                    * bnode -- (default) remove all unreferenced bnodes
                    * all -- remove all unreferenced bnode(s) AND uri(s)
        bnodeCheck -- boolean
                    * True -- (default) check bnodes and raise exception if there are
                              still references to this node
                    * False -- do not check.  This can leave orphaned object reference
                               in triples.  Use only if you are resetting the value in
                               the same transaction
        """
```

The important thing to understand here is that the default behavior is to cascade the delete recursively, deleting all object nodes that are not the object of any other triples. This correctly deletes all lists and containers and things like the maintainer triples for a DOAP record or the author records of a bibliographic item.

CHAPTER

THREE

SPARQL ENDPOINTS

3.1 SPARQLGraph

Warning: early alpha code at work there. Works by providing read-only access.

This module can be imported separately:

```
from rdfalchemy.sparql import SPARQLGraph
```

It is not dependent on the rest of RDFAlchemy and so you can use it as a drop-in replacement for many `rdflib.graph.ConjunctiveGraph` applications.

Ported methods include:

- **`triples()` including derivative methods like:**
  - `subjects()`, `predicates()`, `objects()`
  - `predicate_objects()`, `subject_predicates()`, etc.
  - `value()`

The following update methods will **not** work for SPARQL Endpoints because they are read only (see Sesame below)

- `add()` and `remove()` including derivatives like `set()`
- `parse()` and `load()`, including the ability to load from a url

***SELECT*::** Returns a generator of tuples for each return result

***CONSTRUCT*::**

Returns an rdflib `ConjunctiveGraph('IOMemory')` instance which can be:

- queried through the rdflib api
- assigned as the *db* element to an rdfSubject instance
- serialized to 'n3' or 'rdf/xml'

class `rdfalchemy.sparql.`**`SPARQLGraph`**(*url*, *context=None*)

provides (some) rdflib api via http to a SPARQL endpoint gives 'read-only' access to the graph constructor takes http endpoint and repository name e.g. SPARQLGraph('http://localhost:2020/sparql')

`comment`(*subject*, *default=''*)

Query for the RDFS.comment of the subject

Return default if no comment exists

`construct` (*strOrTriple*, *initBindings={}*, *initNs={}*)

Executes a SPARQL Construct :param strOrTriple: can be either

- a string in which case it it considered a CONSTRUCT query
- a triple in which case it acts as the rdflib *triples((s,p,o))*

Parameters

- **initBindings** – A mapping from a Variable to an RDFLib term (used as initial bindings for SPARQL query)
- **initNs** – A mapping from a namespace prefix to a namespace

Returns an instance of rdflib.ConjuctiveGraph('IOMemory')

`describe` (*s_or_po*, *initBindings={}*, *initNs={}*)

Executes a SPARQL describe of resource

Parameters

- **s_or_po** – is either
  - a subject ... should be a URIRef
  - a tuple of (predicate,object) ... pred should be inverse functional
  - a describe query string
- **initBindings** – A mapping from a Variable to an RDFLib term (used as initial bindings for SPARQL query)
- **initNs** – A mapping from a namespace prefix to a namespace

`items` (*list*)

Generator over all items in the resource specified by list

list is an RDF collection.

`label` (*subject*, *default=''*)

Query for the RDFS.label of the subject

Return default if no label exists

`objects` (*subject=None*, *predicate=None*)

A generator of objects with the given subject and predicate

`predicate_objects` (*subject=None*)

A generator of (predicate, object) tuples for the given subject

`predicates` (*subject=None*, *object=None*)

A generator of predicates with the given subject and object

`qname` (*uri*)

turn uri into a qname given self.namespaces This works for rdflib graphs and is defined for SesameGraph but is **not** part of SPARQLGraph

`query` (*strOrQuery*, *initBindings={}*, *initNs={}*, *resultMethod='xml'*, *processor='sparql'*, *rawResults=False*)

Executes a SPARQL query against this Graph

Parameters

- **strOrQuery** – Is either a string consisting of the SPARQL query

- **initBindings** – *optional* mapping from a Variable to an RDFLib term (used as initial bindings for SPARQL query)
- **initNs** – optional mapping from a namespace prefix to a namespace
- **resultMethod** – results query requested (must be 'xml' or 'json') xml streams over the result set and json must read the entire set to succeed
- **processor** – The kind of RDF query (must be 'sparql' or 'serql')
- **rawResults** – If set to *True*, returns the raw xml or json stream rather than the parsed results.

`subject_objects`(*predicate=None*)
A generator of (subject, object) tuples for the given predicate

`subject_predicates`(*object=None*)
A generator of (subject, predicate) tuples for the given object

`subjects`(*predicate=None*, *object=None*)
A generator of subjects with the given predicate and object

`transitive_objects`(*subject*, *property*, *remember=None*)
Transitively generate objects for the *property* relationship

Generated objects belong to the depth first transitive closure of the *property* relationship starting at *subject*.

`transitive_subjects`(*predicate*, *object*, *remember=None*)
Transitively generate objects for the *property* relationship

Generated objects belong to the depth first transitive closure of the *property* relationship starting at *subject*.

`triples`(*(s, p, o)*, *method='CONSTRUCT'*)

Returns a generator over triples matching the pattern

Parameters **method** – must be 'CONSTRUCT' or 'SELECT'

- CONSTRUCT calls CONSTRUCT query and returns a Graph result
- SELECT calls a SELECT query and returns an interator streaming over the results

Use SELECT if you expect a large result set or may consume less than the entire result

`value`(*subject=None*, *predicate=rdflib.term.URIRef('http://www.w3.org/1999/02/22-rdf-syntax-ns#value')*, *object=None*, *default=None*, *any=True*)
Get a value for a pair of two criteria

Exactly one of subject, predicate, object must be None. Useful if one knows that there may only be one value.

It is one of those situations that occur a lot, hence this *macro* like utility

Parameters

- **predicate, object** (*subject,*) – exactly one must be None
- **default** – value to be returned if no values found
- **any** – if more than one answer return **any one** answer, otherwise *raise UniquenessError*

3.1.1 Sesame endpoints

Can provide read access of Sesame through endpoints. *SELECT* and *CONSTRUCT* methods supported.

If you know you have a Sesame2 endpoint then use the `SesameGraph` rather than `SPARQLGraph` as it has different capabilities.

3.1.2 Joseki endpoints

Can provide read access of Sesame through endpoints. *SELECT*, *CONSTRUCT*, and *DESCRIBE* methods supported.

***triples*::** works but does not currently operate as a true stream. Therefore:

```
db.triples((None,None,None))
```

will attempt to load the entire endpoint into a memory resident graph and **then** iterate over the results.

3.1.3 Relational Data thru SPARQL

In general if your data is in a relational database, you will probably want to use SQLAlchemy as your ORM. If, however that data is in a relational table (yours or someone else's) across the web, and has a SPARQL Mapper on top of it, RDFAlchemy becomes your tool.

D2R Server

D2R Server includes a Joseki servelet. If you depoloy a D2R Server you can access your relational database table through the web as an RDF datastore. RDFAlchemy usage looks like SQLAlchemy but now it can reach across the web into your RDBMS (PostgreSQL, MySQL, Oracle, db2 etc).

3.2 SesameGraph

The RDFAlchemy trunk now includes access to openrdf Sesame2 datastores. SesameGraph is a subclass of SPARQL-Graph and builds on SPARQL endpoint capabilities as it provides write access via a Sesame2 HTTP Protocol.

Just pass the url of the Sesame2 repository endpoint and from there you can use an rdflib type API or use the returned graph in `rdfSubject` as you would any rdflib database.

This module can be imported separately:

```
from rdfalchemy.sparql.sesame2 import SesameGraph
```

it as a drop-in replacement for many `rdflib.graph.ConjunctiveGraph` applications.

Ported methods include:

- **`triples()` including derivative methods like:**
  - `subjects()`, `predicates()`, `objects()`
  - `predicate_objects()`, `subject_predicates()`, etc.
  - `value()`
  - `add()` `remove()` `set()`
  - `parse()` and `load()`, including the ability to load from a url

```
from rdfalchemy.sesame2 import SesameGraph
from rdflib import Namespace

doap = Namespace('http://www.w3.org/1999/02/22-rdf-syntax-ns#doap')
rdf = Namespace('http://www.w3.org/1999/02/22-rdf-syntax-ns#')

db = SesameGraph('http://localhost:8080/sesame/repositories/testdoap')
db.load('data/rdfalchemy_doap.rdf')
db.load('http://doapspace.org/doap/some_important.rdf')

project = db.value(None,doap.name,Literal('rdflib'))
for p,o in db.predicate_objects(project):
   print('%-30s = %s'%(db.qname(p),o))
```

3.2.1 RDFAlchemy use of Sesame

You can use it as you would any rdflib database. Near the head of your code, place a call like

```
from rdfalchemy.sesame2 import SesameGraph
rdfSubject.db = SesameGraph('http://some-place.com/repository')
```

3.2.2 Sesame2 Graph

class `rdfalchemy.sparql.sesame2.`**`SesameGraph`**(*url*, *context=None*)

openrdf-sesame graph via http Uses the sesame2 HTTP communication protocol to provide rdflib type api constructor takes http endpoint and repository name e.g. SesameGraph('http://www.openvest.org:8080/sesame/repositories/Test')

`add`((*s*, *p*, *o*), *context=None*)

Add a triple with optional context

`contexts`

context list ... pretty slow

`get_contexts`()

context list ... pretty slow

`get_namespaces`()

Namespaces dict

`namespaces`

Namespaces dict

`parse`(*source*, *publicID=None*, *format='xml'*, *method='POST'*)

Parse source into Graph

Graph will get loaded into it's own context (sub graph). Format defaults to 'xml' (AKA: rdf/xml).

Returns Returns the context into which the source was parsed.

Parameters

- **source** – source file in the form of "http://....." or "~/dir/file.rdf"
- **publicID** – *optional* the logical URI if it's different from the physical source URI.
- **format** – must be one of 'xml' or 'n3'
- **method** – must be one of

 - 'POST' – method adds data to a context
 - 'PUT' – method replaces data in a context

qname (*uri*)
turn uri into a qname given self.namespaces

query (*strOrQuery*, *initBindings={}*, *initNs={}*, *resultMethod='brtr'*, *processor='sparql'*, *rawResults=False*)
Executes a SPARQL query against this Graph

Parameters

- **strOrQuery** – Is either a string consisting of the SPARQL query
- **initBindings** – *optional* mapping from a Variable to an RDFLib term (used as initial bindings for SPARQL query)
- **initNs** – optional mapping from a namespace prefix to a namespace
- **resultMethod** – results query requested (must be 'xml', 'json' 'brtr') xml streams over the result set and json must read the entire set to succeed
- **processor** – The kind of RDF query (must be 'sparql' or 'serql')
- **rawResults** – If set to *True*, returns the raw xml or json stream rather than the parsed results.

remove (*(s, p, o)*, *context=None*)
Remove a triple from the graph

If the triple does not provide a context attribute, removes the triple from all contexts.

set (*(subject, predicate, object)*)
Convenience method to update the value of object

Remove any existing triples for subject and predicate before adding (subject, predicate, object).

triples (*(s, p, o)*, *context=None*)
Generator over the triple store

Returns triples that match the given triple pattern. If triple pattern does not provide a context, all contexts will be searched.

3.3 Parsers

class `rdfalchemy.sparql.parsers.`**_JSONSPARQLHandler** (*url*)
Parse the results of a sparql query returned as json.

Note: this uses json.load which will consume the entire stream before returning any results. The XML handler uses a generator type return so it returns the first tuple as soon as it's available *without* having to comsume the entire stream

class `rdfalchemy.sparql.parsers.`**_XMLSPARQLHandler** (*url*)
Parse the results of a sparql query returned as xml.

Note: returns a generator so that the first tuple is available as soon as it is sent. This does **not** need to consume the entire results stream before returning results (that's a good thing :-).

class `rdfalchemy.sparql.parsers.`**_BRTRSPARQLHandler** (*url*)
Handler for the sesame binary table format BRTR

CHAPTER

FOUR

ENGINE ROOM

`rdfalchemy.engine.`**`engine_from_config`**(*configuration*, *prefix='rdfalchemy.'*, ***kwargs*)

Create a new Engine instance using a configuration dictionary.

Parameters

- **configuration** – a dictionary, typically produced from a config file where keys are prefixed, such as *rdfalchemy.dburi*, etc.
- **prefix** – indicates the prefix to be searched for.

`rdfalchemy.engine.`**`create_engine`**(*url='‘*, *identifier='‘*, *create=False*)

Returns returns an opened rdflib ConjunctiveGraph

Parameters

- **url** – a string of the url
- **identifier** – URIRef of the default context for writing e.g.:
  - create_engine(‘mysql://myname@localhost/rdflibdb’)
  - create_engine(‘sleepycat://~/working/rdf_db’)
  - create_engine(‘zodb:///var/rdflib/Data.fs’)
  - create_engine(‘zodb://localhost:8672’)
  - create_engine(‘sesame://www.example.com:8080/openrdf-sesame/repositories/Test’)
  - create_engine(‘sparql://www.example.com:2020/sparql’)

for zodb:

the key in the Zope database is hardcoded as ‘rdflib’ urls ending in *.fs* indicate FileStorage otherwise ClientStoreage is assumed which requires a ZEO Server to be running

CHAPTER

FIVE

CUSTOMIZING LITERALS

RDFAlchemy now imports `Literal` from its own file rather than rdflib. This is to provide some customized handling of literals. You can edit your `Literal` file or use the `Literal` source as a model for your own project.

5.1 Literal to Python

For Literals being coverted to Python (i.e. things coming out of the triplestore into your code) rdflib provides a `bind()` method to rebind an XSD datatype to a Python conversion function.

5.1.1 decimal

The problem: Openvest projects originated from the world of finance and investments. Accounting apps especially, need things to add up. That makes the default use of the `float` type troublesome. Take for example:

```
>>> payments=[.1, .1, .1, -.3]
>>> sum(payments) == 0
False
>>> #because
... payments # are floating point numbers like:
[0.10000000000000001, 0.10000000000000001, 0.10000000000000001, -0.29999999999999999]
```

5.2 Literal to Python

The fix here is pretty simple. Just import `Decimal` from the Python `decimal` module and bind it to the datatype.

```
from rdflib import Namespace, Literal
from decimal import Decimal
from rdflib.Literal import bind as bindLiteral

XSD = Namespace(u'http://www.w3.org/2001/XMLSchema#')

bindLiteral(XSD.decimal,Decimal)
```

5.3 Python to Literal

5.3.1 datetime

The problem: Currently (rdflib2.4 on OSX or Linux) cannot round trip a `datetime` Literal. The problem is in the method that parses a string back into a python `datetime` object. It doesn't like microseconds.

```
>>> from rdflib import Literal, Namespace
>>> from datetime import datetime
>>> XSD_NS = Namespace(u'http://www.w3.org/2001/XMLSchema#')
>>> Literal('2008-02-09T10:46:29', datatype=XSD_NS.dateTime).toPython()
datetime.datetime(2008, 2, 9, 10, 46, 29)
>>> # OK that worked but:
... Literal('2008-02-09T10:46:29.234', datatype=XSD_NS.dateTime).toPython() # should return a python
rdflib.Literal('2008-02-09T10:46:29.234', language=None, datatype=rdflib.URIRef('http://www.w3.org/2(
```

5.4 Literal to Python

A better parsing function has been added that you can view `Literal`. This parser will handle microseconds and even slightly mangled iso strings.

This same approach could be used if you prefered to have the `mx.DateTime` moudule and work with `mx.DateTime` instances rather than datetime.

Pick the parser you prefer and perform a binding as shown above for `Decimal`.

CHAPTER

SIX

FORMALCHEMY'S RDFALCHEMY EXTENSION

Auto-generated, customizable HTML input form fields from your RDFAlchemy mapped classes. See FormAlchemy's `support for RDFAlchemy`

"FormAlchemy eliminates boilerplate by autogenerating HTML input fields from a given model. FormAlchemy will try to figure out what kind of HTML code should be returned by introspecting the model's properties and generate ready-to-use HTML code that will fit the developer's application."

CHAPTER
SEVEN

CAPABILITIES

- SQLAlchemy interface
- Caching of data reads
- **Access multiple datastores:**
  - rdflib (beta)
  - **SPARQL endpoints (alpha)**
    - Joseki based Jena access (**alpha**)
    - D2R-server (**alpha**)
  - Access to RDF triples from SQL databases through D2Rq

CHAPTER

EIGHT

NEWS

Trunk now includes:

- Read/Write access for collections and containers
- Read access to SPARQL endpoints
- Read/Write access to Sesame2
- Cascading delete
- chained descriptors and predicate range->class mapping

CHAPTER

NINE

INSTALLATION

RDFAlchemy is now available at Pypi: Just type

```
$ easy_install rdfalchemy
```

If you don’t have setuptools installed...well you should so go get it. Trust me.

CHAPTER

TEN

CODE

Browse dev code at http://bitbucket.org/gjhiggins/rdfalchemy-dev and see the current trunk and all history.

This is an actively developing project so bugs come and go. Get your svn access to the trunk at:

```
$ hg clone http://bitbucket.org/gjhiggins/rdfalchemy-dev
```

CHAPTER

ELEVEN

USER GROUP

You can now visit rdfalchemy-dev at Google Groups.

Bugs can be reported at bitbucket.

CHAPTER

TWELVE

API DOCS

There are epydoc API Docs at http://www.openvest.com/public/docs/rdfalchemy/api/. You can also use links there to browse source, but it might not be current with the trunk.

CHAPTER

THIRTEEN

SAMPLES

There is a samples submodule where some classes like `Foaf` and `Doap` show sample usage of RDFAlchemy and a subdirectory of some rdf Schemes.

CHAPTER

FOURTEEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX